Towards C-arm CT Reconstruction on Larrabee

September 6th 2009
HPIR workshop

Hannes G. Hofmann
B. Keck, C. Rohkohl, J. Hornegger

Chair of Pattern Recognition
Friedrich-Alexander-University Erlangen-Nuremberg
Need for Accelerated Reconstruction?

- Interventional imaging
  - Patient lying on the table
  - → Want results faster

- Less than optimal acquisition
  - E.g. short-scan, motion, irregular samp.
  - → Want better results by use of more complex algorithms
  - Iterative or statistical methods
  - 4-D reconstruction
Outline

RabbitCT Reconstruction Benchmark

Larrabee Architecture

Hands on Code

Conclusion
RabbitCT Reconstruction Benchmark
Why a Standardized Benchmark?

- Different factors affect performance:
  - Number of projection images
  - Volume size & resolution
  - Size of projection images
  - Acquisition geometry

- Every research group uses its own dataset

→ Publications hard to compare
RabbitCT Reconstruction Benchmark

- Dataset and SDK freely available (www.rabbitct.com)
- Well-defined dataset & problem
- SDK: Framework and OpenSource reference implementation
- Accuracy and performance measurement
- Ranking (optional, on the website)

RabbitCT Dataset & Problem Statement

RCT Dataset
- 496 preprocessed projection images
- 1240 x 960 pixel
- Projection matrices from offline calibration
- Reference reconstructions

RCT problem statement
- Backprojection from FDK method
- Different volume size: \{256, 512, 1024\}^3
Larrabee Architecture

- Similar to GPU
  - Add-on card
  - DirectX, OpenGL
  - Fixed-function hardware
  - „High memory bandwidth“ (cmp. to GPUs)
  - Texture sampling units

- But
  - Better understanding of hardware
    - Many independent CPU cores (x86) on one die
  - C/C++ compiler, Intel programming tools
  - Nice logo:
Larrabee Cores

- Based on Pentium design (full x86 support)
  - L1: 32KB
  - L2: 256KB local subset of global coherent L2
  - Added 64-bit support
- Hardware multi-threading (4x SMT)
- Added 512-bit SIMD unit
  - New ISA extension: LRBni (similar to SSE)
  - 32 vector registers
  - Vector-scalar dual issue
Larrabee New Instructions (LRBni)

- Multiply-add
- Load-op: third operand from memory
- Broadcast, swizzle, format conversion
- Gather/scatter
- Predication
Larrabee's Parallel Execution Units

- Many cores – task & data level parallelism
- Wide vector units – data level parallelism
Larrabee Architecture – Bird's Eye

Figure from: Bader: „Game Physics on the Larrabee Architecture“
Hands on Code
Sample Code: Multi-threading

for each projection image {

  for (z=0; z<L; ++z) {
    for (y=0; y<L; ++y) {
      for (x=0; x<L; ++x) {
        process_voxel(x, y, z);
      }
    }
  }
}
Sample Code: Multi-threading

for each projection image {
    #pragma omp parallel for
    for (z=0; z<L; ++z) {
        for (y=0; y<L; ++y) {
            for (x=0; x<L; ++x) {
                process_voxel(x, y, z);
            }
        }
    }
}

- OpenMP and Threading Building Blocks (TBB) will be supported on LRB
- Creating lots of tasks helps hiding latency stalls
SIMD Processing on Larrabee

\[
c = _\text{mm512_add_ps}(a, b);
\]

Does this look familiar?
Sample Code: SIMD Processing

- **Field-of-View Check:** Is voxel visible on projection?
- **Compare projected coordinates with projection size**

```c
// given: __m512i viU, viV, __m512i viZero;
__mmask vmInTop = _mm512_cmpnlt_pi(viV, viZero);   // x4
__mmask vmInV  = _mm512_vkand(vmInTop, vmInBott);  // x2
__mmask vmIn   = _mm512_vkand(vmInU,  vmInV);
```

- **Almost identical to SSE**
SIMD „Problems“ in FDK Code

- Conditional branching complicated
  - Field-of-View check: Is voxel visible on projection?

- Indirect memory access
  - Pixel access: Load 16 pixel values from non-contiguous memory locations
Conditional Branching and SIMD

viU := u-Coordinates of 4 voxels
viZero := \{0, 0, 0, 0\}

One of the comparisons

Result of comparison

Conditional code

SIMD-Problem: Same Instructions for all elements!
LRBni Solution: Predication

\[ c = \text{*_mm512_mask_add_ps(i, m, a, b)*} \]

- Initial value \((i_0 .. i_{15})\) can be any vector
Sample Code: Predication

- **Field-of-View Check:** Is voxel visible on projection?
- **Compare projected coordinates with projection size**
  
  Inside: compute offset

  Outside: set coords to (0,0) to avoid illegal access

```c
// given: __m512i viU, viV, viSu;
// given: __mmask vmIn; __m512i viZero;

__m512i viIdx = _mm512_mask_add_pi( viZero, vmIn, viU, _mm512_mull_pi(viV, viSu) );
```

* Initial value, Mask
Indirect Memory Access and SIMD

Projection data

\( \text{vIdx}: \text{Pixel indices / -offsets} \)

\( \text{vfValues}: \text{Pixel values} \)

SIMD-Problem: Vectors have to be loaded contiguously!
Sample Code: Gather/Scatter

- Pixel Access: Load 16 pixel values from non-contiguous memory locations

```c
// given: float* pProj;
// given: __mmask vmIn; __m512i viIdx; __m512 vfZero;
const int upConv = _MM_FULLUPC_NONE; // no up conversion
const int scale = sizeof(float); // element size
__m512 vfValues = _mm512_vgatherd_loop( vfZero, vmIn, viIdx, pProj, upConv, scale );
```

* Initial value, Mask

- Also handy for AOS ↔ SOA conversions
Texture Samplers

- Texture Samplers run asynchronous

- Interesting for reconstruction: Interpolation
  - Support more functions (DirectX)

- Essential for peak performance
Porting Code to Larrabee (1)

- Existing, optimized CPU implementations (OpenMP or TBB, SSE)

- Host program
  - Calls LRB functions (cmp. CUDA host code)

- Multi-threading libraries are supported
Porting Code to Larrabee (2)

- **SSE mapping straightforward**
  1) Replace data types
  2) Adjust number of loop iterations
  3) Replace SSE instructions by LRBni

- Then, optional: LRB Optimizations
Conclusion

- Larrabee is well suited for reconstruction
  (for any other task as well, probably)

- Predication

- Gather/scatter

- Easy porting of legacy code (even optimized)
  - It's still x86 for the 90% of code that just has to work
  - LRBni similar to SSE
  - Support for OpenMP, TBB
Thanks for Your Attention

- RabbitCT Reconstruction Benchmark
- Larrabee architecture
- Sample Code
  - Multi-threading, SIMD, Predication (Masks), Gather/Scatter
- Porting legacy code to LRB
- Summary

hannes.hofmann@cs.fau.de
http://www5.cs.fau.de/~hofmann
Backup Slides
if (u >= 0 && u < Su && v >= 0 && v < Sv) // branch
{
    int idx = v * Su + u;              // idx
    return pProj[ idx ];              // load
}
return 0.0;                           // default
// Note: __mm_cmp* insns:
//       true=>0xFFFFFFFF, false=>0x00000000
__m128i vmInTop = __mm_cmpgt_epi32(viV, viZero); // x4
__m128i vmInV   = __mm_and_si128(vmInTop, vmInBott); // x2
__m128i vmIn    = __mm_and_si128(vmInU, vmInV);
__m128i viIdx   = __mm_add_epi32( viU,
    __mm_mullo_epi32(viV, viSu) );
__m128i viIdxMasked = __mm_and_si128(viIdx, vmIn);
// Skipped: Load elements sequentially into vfValues
__m128  vfValMasked = __mm_and_si128(vfValues, vmIn);
return vfValMasked;

(Simple case, because default = 0x00000000)
LRBni Code: Load 16 Pixels

```c
const int upConv = _MM_FULLUPC_NONE; // no up conversion
const int scale = sizeof(float); // element size

__mmask vmInTop = _mm512_cmpnlt_pi(viV, viZero); // x4
__mmask vmInV = _mm512_vkand(vmInTop, vmInBott); // x2
__mmask vmIn = _mm512_vkand(vmInU, vmInV);

__m512i viIdx = _mm512_mask_add_pi(viZero, vmIn, viU,
    _mm512_mull_pi(viV, viSu));

__m512 vfValues = _mm512_vgatherd_loop(vfZero, vmIn,
    viIdx, pProj, upConv, scale);

return vfValues;
```

(Initial value, Mask)
Pseudo-Code

For each projection \((n = 0..N-1)\)
For each Voxel \((z,y,x = 0..L-1)\)
    Project voxel onto image plane
    Sample projection image
    Accumulate voxel value
HPC Challenges

- Exploit parallel execution units
  - Within a core: Vectorization (SIMD)
  - Many cores: Multi-threading

- Reduce memory transfers
  - Many algorithms are limited by memory bandwidth
  - Use caches efficiently – keep data close to the cores

- Exploit specialized hardware features
  - E.g. texture samplers, gather/scatter unit, cache control, etc.
Furthermore...

- Fixed-Function Hardware
  - Hm... Gather/Scatter Engine
- Befehle zur Cache-Steuerung
  - Prefetch, Write-Through